AMENDMENTS TO THE SPECIFICATION

Please amend paragraph [0044] as follows:

[0044] In the course of translating 151 the first basic block of subject code 153 into target code, the translator 30 generates an IR tree 163 based on that basic block. In this case, the IR tree 163 is generated from the source instruction "add%eex, %edx," which is a flag-affecting instruction. In the course of generating the IR tree 163, four abstract registers are defined by this instruction: the destination subject register %ecx 167, the first flag-affecting instruction parameter 169, the second flag-affecting instruction parameter 171, and the flag-affecting instruction result 173. The[[,]] IR tree corresponding to the "add" instruction is ~~simple~~ simply a 'Y' (arithmetic addition) operator 175, whose operands are the subject registers %ecx 177 and %edx 179.

Please amend paragraph [0045] as follows:

[0045] Emulation of the first basic block puts the flags in a pending state by storing the parameters and result of the flag-affecting instruction. ~~The flag-affecting instruction is "add %eex, %edx."~~ The flag-affecting instruction is "add %ecx, %edx". The parameters of the instruction are the current values of emulated subject registers %ecx 177 and %edx 179. The "@" symbol preceding the subject register uses 177, 179 indicate that the values of the subject registers are retrieved from the global register store, from the locations corresponding to %ecx and %edx, respectively, as these particular subject registers were not previously loaded by the current basic block. These parameter values are then stored in the first 169 and second 171 flag parameter abstract registers. The result of the addition operation 175 is stored in the flag result abstract register 173.

Please amend paragraph [0047] as follows:

[0047] In the course of translating 157 the second basic block of subject code 159, the translator 30 generates an IR tree 165 based on that basic block. The IR tree 165 is generated from the source instruction ~~"pushf,"~~ "pushf", which is a flag-using instruction. The semantics of the "pushf" instruction are to store the values of all condition flags onto the stack, which requires that each flag be explicitly calculated. As such, the abstract registers corresponding to four condition flag values are defined during IR generation: the zero flag ("ZF") 181, the sign flag ~~("S17")~~ ("SF") 183, the carry flag ("CF") 185, and the overflow flag ("OF") 187. Node 195 is the arithmetic comparison operator ~~"unsigned less-than."~~ "unsigned less-than". The calculation of the condition flags is based on information from the prior flag-affecting instruction, which in this case is the "add %ecx, %edx" instruction from the first basic block 153. The IR calculating the condition flag values 165 is based on the result 189 and parameters 191, 193 of the flag-affecting instruction. As above, the "1" symbol preceding the flag parameter labels indicates that the emulator inserts target code to load those values from the global register store prior to their use.

Please amend paragraph [0059] as follows:

[0059] Complex nodes reduce the need for idiom recognition in the backend 33, because complex nodes contain more semantic information than their base node equivalents. Specifically, complex nodes avoid the need for backend 33 idiom recognition of constant operands. By comparison, if an immediate type subject instruction were decomposed into base nodes (and the target architecture also contained immediate type instructions), then the translator 30 would either need expensive backend 33 idiom recognition to identify the multiple node cluster as an immediate instruction candidate, or generate inefficient target code (i.e., more instructions than necessary, using more target registers than ~~necessary.~~ necessary). In other words, by utilizing base nodes alone, performance is lost either in the translator 30 (through idiom recognition) or the translated code (through extra generated code without idiom recognition). More generally, because complex nodes are a more compact representation of semantic information, they reduce the number of IR nodes that the translator 30 must create, traverse, and delete.

Please amend paragraph [0060] as follows:

[0060] Immediate type instructions are common to many architectures. Therefore, complex nodes are generic in that they are reusable across a range of architectures. However, not every complex node is present in the IR node set of every translator. Certain generic features of the translator are configurable, meaning that when a translator is being compiled for a particular pair of source and target architectures, features that do not apply to that translator configuration can be excluded from compilation. For example, in a MIPS-MIPS (MIPS to MIPS) translator, complex nodes that do not match the semantics of any MIPS instructions are excluded from the IR node set because they would never be utilized.

Please amend paragraph [0061] as follows:

[0061] Complex nodes can further improve the performance of the target code generated using an in-order traversal. In-order traversal is one of several alternative IR traversal algorithms that determines the order in which IR nodes within an IR tree are generated into target code. Specifically, in-order traversal generates target code for each IR node as it is first traversed, which precludes backend 33 idiom recognition due to the absence of a separate optimization pass over the entire IR tree. Complex nodes represent more semantic information per node than base nodes, and thus some of the work of idiom recognition is implicit within the complex nodes themselves. This allows the translator 30 to use in order traversal without suffering much of a penalty in target code performance as it would with base nodes alone.

Please amend paragraph [0071] as follows:

[0071] To illustrate the process of optimizing the translator 30 for utilizing polymorphic nodes in the IR, the following example describes the translation of a PPC (PowerPC "SHL64" instruction (left shift, 64 bit) required in a PPC-P4 (PowerPC to Pentium4) translator using first base nodes and then polymorphic nodes.

Please amend paragraph [0071] as follows:

[0071] The frontend decoder 200 of an unoptimized translator decodes the current block and encounters the PPC SHL64 instruction. Next, the frontend realize block 202 instructs the kernel 32 to construct an IR consisting of multiple base nodes. Then the kernel 32 optimizes the IR forest (generated from the current block of instructions) and performs an in-order~~ing~~ traversal ~~to determine the order of code generation in Base IR block 204~~. Next, the kernel 32 performs code generation for each IR node ~~in order~~, instructing the backend 33 to plant appropriate RISC type instructions. Finally, the backend 33 plants code in plant block 206 and encodes each RISC type instruction with one or more target architecture instructions in encode block 208.

Please amend paragraph [0076] as follows:

[0076] PPC SHL64 => Poly IR single node => P4 single/few instructions

Please amend paragraph [0077] as follows:

[0077] The frontend decoder 200 of the optimized translator 30 decodes the current block and encounters the PPC SHL6instruction. Next, the frontend realize block 202 instructs the kernel 32 to construct an IR consisting of a single polymorphic IR node. When the single polymorphic node is created, the backend 33 knows that the shift operand of SHL6must be in a specific register (%ecx on P4). This requirement is encoded in the polymorphic node. Then the kernel 32 optimizes the IR forest for current block and performs an in-order~~ing~~ traversal ~~to fix the code generation order in the polymorphic IR block 212~~. Next, the kernel 32 performs code generation for each node, instructing the backend 33 to plant appropriate RISC type instructions. During code generation, however, polymorphic nodes are treated differently than base nodes. Each polymorphic node causes the invocation of a specialized code generator function which resides in the backend 33. The backend 33 specialized code generator function plants code in plant block 216 and encodes each subject architecture instruction with one or more target architecture instructions in encode block 208. During register allocation in the generation phase, the specific register information is used to allocate the correct register. This reduces the computation performed by the backend 33 which would be required if unsuitable registers had been allocated. This code generation may involve register allocation for temporary registers.

Please amend paragraph [0088] as follows:

[0088] The implementation component of an ASN is specific to the node's architecture, such that it generates an architecture specific instruction corresponding to that ~~ASK~~ ASN. For example, the implementation component of a MIPSLoad ASN generates a MIPS "Id" (load) instruction. When using the translator of the present invention with the same subject and target architectures (i.e., as an accelerator), subject ASNs will possess implementation components. When utilizing the translator with different subject and target architectures, subject ASNs will have conversion components.

Please amend paragraph [0089] as follows:

[0089] For example, FIG. 7 illustrates the ASN for a MIPS instruction when using an embodiment of the present invention in a MIPS-MIPS accelerator. The frontend 31 decodes the MIPS "addi" (add immediate) instruction 701 and generates an IR to include the corresponding ASN, MIPS_ADDI 703. The subject and target architectures are the same for an accelerator, and thus the conversion component "CVT" 707 is undefined. The implementation component ~~"IMPL~~ "IMPL" 705 is defined to generate the same MIPS "addi" instruction 709, subject to register allocation differences in the code generation pass.

Please amend paragraph [0090] as follows:

[0090] FIG. 8 illustrates the ASNs in the IR for the same MIPS instruction when using an embodiment of the present invention in a ~~MIPS-X86~~ MIPS-X86 translator. The frontend 31 decodes the MIPS "addi" subject instruction and generates a corresponding subject ASN, ~~MIPS ADDI~~ MIPS_ADDI 801. The source and target architectures are different for this translator, and the implementation component 803 of the subject ASN 801 is thus undefined. The conversion component 805 of the ~~MIPS-ADDI~~ MIPS_ADDI is a specialized conversion component, which converts the subject ASN 801 into a target ASN 807. By comparison, a generic conversion component would convert the subject ASN 801 into a base node representation. The target ASN representation of the MIPS ADDI node 801 is a single X86 ADDI node 807. The conversion component 811 of the target ASN 807 is undefined. The implementation component 809 of the

target ASN 807 generates the a target instruction 813, in this case the X86 instruction "ADD $EAX, #10."

Please amend paragraph [0103] as follows:

[0103] Referring now to FIGS. 10, 11 and 12, examples illustrating the different IR trees that are generated from the same MIPS instruction sequence using base IR nodes, MIPS-MIPS ASN IR nodes, and MIPS-X86 ASN IR nodes, respectively, are shown. The semantics of the example MIPS subject instruction sequence (load upper immediate, then bitwise-or immediate) is to load the 32 bit constant value ~~0x12345678~~ 0x12345678into subject register ~~"a1 ""~~"a1".

Please amend paragraph [0104] as follows:

[0104] In FIG. 10, the Binary Decoder 300 is a frontend 31 component of the translator 30 which decodes (parses) the subject code into individual subject instructions. After the subject instructions are decoded, they are realized as base nodes 302 and added to the working IR forest for the current block of instructions. The IR Manager 304 is the portion of the translator 30 that holds the working IR forest during IR generation. The IR Manager 304 consists of abstract registers and their associated IR trees (the roots of the IR forest are abstract registers). For example, in FIG. 10, the abstract register ~~"a-V-3-06-~~"a1" 306 is the root of an IR tree 308 of five nodes, which is part of the current block's working IR forest. In a translator 30 implemented in C++, the IR ~~Manager.~~ Manager 304 may be implemented as a C++ object that includes a set of abstract register objects (or references to IR node objects).

Please amend paragraph [0105] as follows:

[0105] FIG. 10 illustrates an IR tree 308 generated by a MIPS to X86 translator using base nodes only. The ~~MIPS_LUI~~ MIPS_LUI instruction 310 realizes a "SHL" (shift left) base node 314 with two operand nodes 316 and 318, which in this case are both constants. The semantics of the MIPS_LUI instruction 310 are to shift a constant value (0xI234) left by a constant number of bits (16). The MIPS_ORI instruction 312 realizes an "ORI" (~~bitwise-or~~ bitwise-or immediate) base node 320 with two operand nodes 314 and 322, the result of the SHL

node 314 and a constant value. The semantics of the MIPS_ORI instruction 312 are to perform a bitwise-or of the existing register contents with a constant value (Ox5678).

Please amend paragraph [0106] as follows:

[0106] In an unoptimized code generator, the base nodes include no immediate-type operators other than ~~load-immediate~~load-immediate, so each constant node results in the generation of a load immediate instruction. The unoptimized base node translator therefore requires five RISC type operations (load, load, shift, load, or) for this subject instructions sequence. Backend 33 idiom recognition can reduce this number from five to two, by coalescing the constant nodes with their parent nodes, to generate immediate type target instructions (i.e., shift immediate and or immediate). This reduces the number of target instructions to two, but for an increased translation cost in performing the idiom recognition in the code generator.

Please amend paragraph [0108] as follows:

[0108] FIG. 11 illustrates the IR tree generated by a ~~MIPS-X86~~ MIPS-X86 (MIPS to X86) translator using ASNs. After the subject instructions are decoded by the binary decoder 300, they are realized as MIPS_X86 ASN nodes 330, which are then added to the working IR forest for the current block. First, the MIPS_X86_LUI ASN node is converted into an X86 32-bit constant node 332 by the ASN's convert component. Second, the MIPS_X86_ORI ASN node produces an X86 ORI node which is immediately folded with the previous X86 constant node (constant folding), resulting in a single X86 32-bit constant node 334. This node 334 is encoded into a single X86 load constant instruction, "mov %eax, $Ox 12345678". As can be seen, ASN nodes result in fewer nodes than the base node example, thus reducing translation cost and providing better target code.

Please amend paragraph [0111] as follows:

[0111] In FIG. 12, the first subject instruction 310 is ~~"lui al, Oxl234".~~ "lui a1, Ox1234". The semantics of this instruction 310 are to load the constant value ~~Oxl234~~ Ox1234 into the upper 16 bits of subject register "al" 342. This instruction 310 realizes a MIPS_MIPS_LUI node 344, with an immediate field constant value of ~~Oxl234.~~ Ox1234. The translator adds this node

to the working IR forest by setting abstract register "al" 342 (the destination register of the subject instruction) to point to the MIPS_MIPS_LUI IR node 344.

[0112] In the same example in FIG. 12, the second subject instruction 312 is ~~44ori al, al, Ox5678".~~"ori al, al, Ox5678". The semantics of this instruction 312 are to perform a bitwise-or of the constant value Ox5678 with the current contents of subject register [[4]]"al" 342 and to store the result in subject register [[4]]"al" 346. This instruction 312 realizes a MIPS_MIPS_ORI node 348, with an immediate field constant value of Ox5678. The translator adds this node to the working IR forest by first setting the ORI node to point to the IR tree that is currently pointed to by abstract register "al" 342 (the source register of the subject instruction), and then setting the abstract register "al" 346 (the ~~destination,~~ destination register of the subject instruction) to point to the ORI node 348. In other words, the existing "al" tree rooted with abstract register 342 (i.e., the LUI node) becomes a subtree 350 of the ORI node 348, and then the ORI node 348 becomes the ~~new al tree.~~ new "al" tree. The old "al" tree (after LUI but before ORI) is rooted from abstract register 342 and shown as linked by line 345, while the current "al" tree (after ORI) is rooted from abstract register 346.